9.5 Exceptions as a Part of the Public Interface

The stack implementations we studied in the previous section included a precondition on their `pop` method specifying that the stack must not be empty. Preconditions are used to rule out erroneous situations like attempting to remove an item from an empty stack, but they come with one drawback: every precondition we add increases the complexity of the function's interface. A precondition becomes the responsibility of the *user* of the function to check, for example, with code like

```python
if not my_stack.is_empty():
    top_item = my_stack.pop()
```

Sometimes these checks are straightforward, but depending on the preconditions we specify, they can be onerous as well. In this section, we'll introduce an alternate mechanism for signaling an erroneous state from within a function call.

Warm-up: letting an error happen

Consider this version of `Stack.pop`, which removes the precondition but keeps the same implementation:

```python
    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.
        """
        return self._items.pop()
```

When we call `pop` on an empty stack, we encounter the following error:

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "...", line 58, in pop
    return self._items.pop()
IndexError: pop from empty list
```

As we saw earlier in the course, when an exception is raised Python stops the normal control flow of the currently running program. From the perspective of the client code, it is good to see an exception to know that something has gone wrong, but bad that the exceptions report refers to a list (`IndexError: pop from empty list`) and a private attribute (`self._items`) that the client code should have no knowledge of.

Custom exceptions

A better solution is to raise a custom exception that is descriptive, yet does not reveal any implementation details. We can achieve this very easily in Python: we define our own type of error by defining a new class:

```python
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""
```

There is some slightly new syntax here: the `(Exception)` that follows the class name. For now, it is enough to know that this will properly create a new type of exception. The technical mechanism used, *inheritance*, is one we'll cover later in this chapter.

Here's how we'll use `EmptyStackError` in our `pop` method:

```python
    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Raise an EmptyStackError if this stack is empty.
        """
        if self.is_empty():
            raise EmptyStackError
        else:
            return self._items.pop()
```

There are two important changes in this version of `pop`. First, in the method docstring there is a new sentence which names both the type of exception and the scenario that will cause that exception to be raised. This exception is now part of the *public interface* of `Stack.pop`, meaning users of this class will be expected to take note of this exception. Second, this implementation now uses a new Python keyword, `raise`, which unsurprisingly raises an exception.[1] A raise statement can be used anywhere in our code to raise exceptions, even

[1] Even though we're using our custom exception class here, `raise` works with any exception type, such as `IndexError` and `AttributeError`.

ones that we've defined ourselves. Let's see what happens now when we call `pop` on an empty stack:

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "...", line 60, in pop
    raise EmptyStackError
EmptyStackError
```

As before, an exception is raised. But now the line shown is just this simple `raise` statement; it doesn't mention any implementation details of the class. And it specifies that an `EmptyStackError` was the problem, as was documented in the method docstring.

Custom exception messages

One current limitation of the above approach is that simply the name of the exception class does not convey a lot of meaning. To provide a custom exception message, we can define a new special method with the name `__str__` in our exception class:[2]

[2] Like `__init__`, the name `__str__` has special meaning in Python. We'll study it and more methods like it later in the course.

```python
class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""

    def __str__(self) -> str:
        """Return a string representation of this error."""
        return 'pop may not be called on an empty stack'
```

```
>>> s = Stack()
>>> s.pop()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "...", line 60, in pop
    raise EmptyStackError
EmptyStackError: pop may not be called on an empty stack
```

Testing exceptions

Because we include `EmptyStackError` as part of the public interface of the `Stack.pop` method, we should write tests to check that this behaviour occurs as expected. But unlike the tests we've written so far, we cannot simply call `pop` on an empty stack and check the return value or the state of the stack after `pop` returns. Raising an error interrupts the regular control flow of a Python program—and this includes test cases!

The `pytest` module[3] allows us to write tests that expects an exception to occur using a function `pytest.raises` together with the `with` keyword: Here is an example of a test case to

[3] It is also possible to write doctests that check for exceptions. See Appendix B.1 for details.

check that calling `Stack.pop` on an empty stack raises an `EmptyStackError`.

```python
# Assuming our stack implementation is contained in a file stack.py.
from stack import Stack, EmptyStackError
import pytest


def test_empty_stack_error():
    """Test that popping from an empty stack raises an exception."""
    s = Stack()

    with pytest.raises(EmptyStackError):
        s.pop()
```

The `with` keyword acts as an assertion, expecting an `EmptyStackError` to be raised by the body of the `with` block, the function call `s.pop()`. The test *passes* when that exception is raised, and *fails* when that exception is not raised (this includes the case when a different exception is raised instead of the expected one).

Handling exceptions

We've said repeatedly that when an exception is raised, the normal execution of the program is stopped, and the exception is reported to the user. However, `pytest.raises` seems to circumvent this: after an `EmptyStackError` is raised in our test, the test simply *passes* and execution proceeds to the next test. How does `pytest.raises` achieve this?

Python provides a compound statement, the **try-except** statement, to execute a block of code and handle a case where one or more pre-specified exceptions are raised in that block. Here is the simplest form of a try-except statement:

```
try:
    <statement>
    ...
except <ExceptionClass>:
    <statement>
    ...
```

When a try-except statement is executed:

1. First, the block of code indented within the `try` is executed.

2. If no exception occurs when executing this block, the `except` part is skipped, and the Python interpreter continues to the next statement after the try-except.

3. If an exception occurs when executing this block:

    - If the exception has type `<ExceptionClass>`, the block under the `except` is executed, and then after that the Python interpreter continues executing the next statement after the try-except.

      Importantly, in this case the program does *not* immediately halt!

    - However, if the exception is a different type, this does stop the normal program execution.

In practice, client code often uses try-except statements to call functions that may raise an error as part of their public interface. This shields users from seeing errors that they should never see, and allows the rest of the program to continue.

For example, here is how we could implement a function that takes a stack and returns the second item from the top of the stack.

```python
def second_from_top(s: Stack) -> Optional[str]:
    """Return the item that is second from the top of s.

    If there is no such item in the Stack, returns None.
    """
    try:
        hold1 = s.pop()
    except EmptyStackError:
        # In this case, s is empty. We can return None.
        return None

    try:
        hold2 = s.pop()
    except EmptyStackError:
        # In this case, s had only one element.
        # We restore s to its original state and return None.
        s.push(hold1)
        return None

    # If we reach this point, both of the previous s.pop() calls succeeded.
    # In this case, we restore s to its original state and return the second
         item.
    s.push(hold2)
    s.push(hold1)

    return hold2
```

References

- CSC108 videos: Exceptions (video)